Developing QuickTime Components 101

Session 605

Developing QuickTime Components 101

Gary Woodcock
Discreet, Inc.

Introduction

- This session covers the concepts common to writing all types of QuickTime components
- We will not focus on the details of writing a specific component type (e.g., an image compressor)

What You Will Learn

- How to leverage QuickTime to deploy your own unique technologies
- How to use components as a extensibility and/or reusability mechanism for your software

You Should Be Familiar With . . .

- The C programming language
- The Memory Manager
- The Resource Manager
- The Component Manager

Session Overview

- What is a component?
- Why should I write a component?
- How do I write and debug a component?
- How do I create my own kind of component?
- How do I deliver my component?

What Is a Component?

- A component is just another kind of code library
- It can be used by multiple clients (reusability)
- It can be private or public (scope)
- It can be versioned

That Is Not Too Unique . . .

- True, but components are interesting because they can:
  - Be searched by capability as opposed to only name or symbol
  - Directly extend the capabilities of QuickTime
  - Be delivered on Mac OS 7.x–9.x, Mac OS X, and Windows

What Else Can They Do?

- Components can “inherit” functionality from other components
- Components can expose user interface
- Components can use other components to create chains of functionality

Libraries Compared

Type	Search by Name?
Component	*Yes*
Static library	*Not applicable*
CFM library	*Yes*
Mach-O library	*Yes*
Windows DLL	*Yes*
Windows COM	*Yes*

Libraries Compared

Type	**Search by Capability?**
Component	*Yes*
Static library	*Not applicable*
CFM library	*No*
Mach-O library	*No*
Windows DLL	*No*
Windows COM	*No*

Libraries Compared

Type	**Dynamically Loaded?**
Component	*Windows, Mac OS X only*
Static library	*No*
CFM library	*Yes*
Mach-O library	*Yes*
Windows DLL	*Yes*
Windows COM	*Yes*

Libraries Compared

Type	**Languages**
Component	*C,C++,Pascal*
Static library	*C,C++,Obj-C,Pascal,etc.*
CFM library	*C,C++,Pascal,etc.*
Mach-O library	*C,C++,Obj-C*
Windows DLL	*C,C++,Pascal,etc.*
Windows COM	*C,C++,C#,VB,etc.*

Libraries Compared

Type	**Supported OS’s**
Component	*Mac OS 9.x, X, Windows*
Static library	*Any*
CFM library	*Mac OS 9.x, X.*
Mach-O library	*Mac OS X*
Windows DLL	*Windows.*
Windows COM	*Windows.*

Popular Components

- Image compressors/decompressors (codecs)
- Movie importers/exporters
- Graphics importers/exporters
- Video digitizers
- Media handlers

A Component Contains . . .

- A dispatcher
- The four required component routines
- Any of the optional component routines
- Routines defined by the component type
- Support resources

Component Manager Review

- The Component Manager manages components and their client connections
- Integral service of Mac OS; on Windows, delivered as part of QuickTime
- Components are identified by type, subtype, and manufacturer

Component Manager (Cont.)

- Components must be registered before they can be used
- Each component type has its own unique API

Component Manager (Cont.)

- Finding a component:

```
ComponentDescription   cd;
Component              compID;

cd.componentType = kMathComponentType;
cd.componentSubtype = kAnyComponentSubtype;
cd.componentManufacturer =
    kAnyComponentManufacturer;
cd.componentFlags = 0L;
cd.componentFlagsMask =
    kAnyComponentFlagsMask;

compID = FindNextComponent (NULL, &cd);
```

Component Manager (Cont.)

- Opening a component:

```
Component              compID;
ComponentInstance      ci;
OSErr                  myErr;

myErr = OpenAComponent (compID, &ci);
```

Component Manager (Cont.)

- Querying a component for supported routines:

```
ComponentInstance       ci;
OSErr                   myResult;
Boolean                 supported;

myResult = CallComponentCanDo (ci,
    kComponentTargetSelect);
if (myResult == 0)
    supported = false;
else
    supported = true;
```

Component Manager (Cont.)

- Getting a component’s version:

```
ComponentInstance       ci;
long                    version;
long                    implementationVersion;
long                    interfaceVersion;

version = CallComponentVersion (ci);
interfaceVersion = (version >> 16L) & 0x0000FFFF;
implementationVersion = (version & 0x0000FFFF);
```

Component Manager (Cont.)

- Targeting a component:

```
ComponentInstance       ci;
ComponentInstance       target;
OSErr                   myErr;

myErr = CallComponentTarget (ci, target);
```

Component Manager (Cont.)

- Closing a component:

```
ComponentInstance       ci;
OSErr                   myErr;

myErr = CloseComponent (ci);
```

Platform Differences

- Under Mac OS 7.x–9.x, there is a global registration list; components can also be registered local to an application process
- Under Mac OS X and Windows, the registration list is local to each process

Platform Differences

- Component reference constants are shared between process address spaces only on Mac OS 7.x–9.x
- Modification seeds apply only on a per-process basis for Mac OS X and Windows
- Instances can be counted only on a per-process basis for Mac OS X and Windows

Platform Differences

- Carbon components do not run under Mac OS 7.x–9.x
- On Mac OS X and Windows, the Register and Unregister routines are invoked each time a new QuickTime process begins

Implications

- Avoid using the component refcon to store common instance data across processes
- Avoid using Register and Unregister as “execute once and once only” setup and teardown
- Avoid counting instances to limit connections to your component

Development Tools

- Project Builder (Mac OS X)
- CodeWarrior (Mac OS 7.x–9.x)
- Visual Studio .NET (Windows)

Can We Start Already?

- We are going to define a component that can add two numbers and produce the result
- It supports the required routines and most of the optional routines
- We will see how calls in the client program translate to routines in the component

Defining an API

- We want to add two numbers:

```
enum {kMathAddSelect = 0};

EXTERN_API (ComponentResult) MathAdd
    (ComponentInstance ci, SInt16 inFirstNum,
    SInt16 inSecondNum, SInt32 * outResult)
    ComponentCallNow (kMathAddSelect,
    sizeof (SInt16) + sizeof (SInt16) +
    sizeof (SInt32 *));
```

Constants

- We need constants for registering and finding our component:

```
#define kTheComponentType               'MATH'
#define kTheComponentSubType            'WWDC'
#define kTheComponentManufacturer'Appl'
#define kTheComponentResID              128
#define kTheComponentAPIVersion     0x00010000
#define kTheComponentImplementationVersion
     0x00000001
```

Globals

- Each component instance needs global storage:

```
struct MathGlobals {
    ComponentInstance   self;
    ComponentInstance   target;
};
typedef struct MathGlobals MathGlobals;
typedef struct MathGlobals * MathGlobalsPtr;
```

The Dispatcher

- The dispatcher is responsible for mapping selectors to routines; You can:
  - Write it yourself
  - Generate it automatically using the **ComponentDispatchHelper.c** file
- It is far simpler to use the latter method!

Dispatch Helper

- **ComponentDispatchHelper.c** is part of the QuickTime SDK
- We will write a dispatch header file that provides values for the #defines required by the dispatch helper

Dispatch Header

- The first portion of our dispatch header looks like this:

ComponentSelectorOffset	(10)
ComponentRangeCount	(2)
ComponentRangeShift	(8)
ComponentRangeMask	(FF)
ComponentStorageType	(Ptr)

Dispatch Header (Cont.)

- This line indicates the number of base selectors specified:

```
ComponentSelectorOffset        (10)
```

- Note this number is the contiguous number of base selectors specified, not the total possible number

Dispatch Header (Cont.)

- This line indicates the number of selector ranges specified:

  ComponentRangeCount (1)

- A selector range is a contiguous set of selectors, where the first selector and the number of selectors are defined by **ComponentRangeShift** and **ComponentRangeMask**

Dispatch Header (Cont.)

- The **ComponentRangeShift** is the amount a selector value is shifted left, then incremented, to produce its range number:

  ComponentRangeShift (8)

- For example, a selector value of 0x0101, using the range shift above, belongs to selector range 2 (1 + 1)

Dispatch Header (Cont.)

- The **ComponentRangeMask** is used to mask the selector value to define which routine entry in a range is mapped to the selector:

  ComponentRangeMask (FF)

- Using the previous example, a selector value of 0x0101 using this mask maps to routine entry 1

Dispatch Header (Cont.)

- This line indicates the type of storage used by our component:

```
ComponentStorageType        (Ptr)
```

- Or alternatively,

```
ComponentStorageType        (Handle)
```

Dispatch Header (Cont.)

- Our first range is defined as follows:

```
ComponentRangeBegin (0)
    ComponentError           (GetPublicResource)
    ComponentError           (ExecuteWiredAction)
    ComponentError           (GetMPWorkFunction)
    StdComponentCall         (Unregister)
    StdComponentCall         (Target)
    StdComponentCall         (Register)
    StdComponentCall         (Version)
    StdComponentCall         (CanDo)
    StdComponentCall         (Close)
    StdComponentCall         (Open)
ComponentRangeEnd (0)
```

Dispatch Header (Cont.)

- And the second range is:

```
ComponentRangeBegin (1)
    ComponentCall (MathAdd)
ComponentRangeEnd (1)
```

Helper #Defines

- We need to set a number of #defines for **ComponentDispatchHelper.c:**

```
#define CALLCOMPONENT_BASENAME()   __Math
#define CALLCOMPONENT_GLOBALS()     \
     MathGlobalsPtr storage
#define COMPONENT_UPP_PREFIX()      uppMath
#define COMPONENT_DISPATCH_FILE     \
     “MathComponentDispatch.h”
#define COMPONENT_SELECT_PREFIX     kMath
#define MATH_BASENAME()\
     CALLCOMPONENT_BASENAME()
#define MATH_GLOBALS()   \
     CALLCOMPONENT_GLOBALS()
```

Helper #Defines (Cont.)

- This #define specifies the common base prefix of the routine names in our component implementation:

```
#define CALLCOMPONENT_BASENAME()    __Math
```

- For example, our component’s Open routine is called “__MathOpen”

Helper #Defines (Cont.)

- This #define specifies the type of instance storage in our component implementation:

```
#define CALLCOMPONENT_GLOBALS()       \
    MathGlobalsPtr storage
```

- Our component instance storage is of type "**MathGlobalsPtr**"

Helper #Defines (Cont.)

- This #define specifies the prefix of the universal procedure pointer constants used by our component:

```
#define COMPONENT_UPP_PREFIX() uppMath
```

Helper #Defines (Cont.)

- This #define specifies the name of the dispatch file used by our component:

```
#define COMPONENT_DISPATCH_FILE()      \
     “MathComponentDispatch.h”
```

Helper #Defines (Cont.)

- Because the same instance storage type and selector prefix is used for both the base routines and our component API routines, we can use the following shortcut:

```
#define MATH_BASENAME()     \
    CALLCOMPONENT_BASENAME()
#define MATH_GLOBALS()      \
    CALLCOMPONENT_GLOBALS()
```

Helper #Includes

- These #includes, together with the #defines we set earlier, generate the dispatcher when compiled:

```
#include “MathComponentSelectors.h”
#include <Components.k.h>
#include “MathComponentSelectors.k.h”
#include <ComponentDispatchHelper.c>
```

- Next, we will see how to write the **MathComponentSelectors.k.h** file

Selectors.k.h

- These are utility macros required by **ComponentDispatchHelper.c:**

```
#ifdef MATH_BASENAME
    #ifndef MATH_GLOBALS
        #define MATH_GLOBALS()
        #define ADD_MATH_COMMA
    #else
        #define ADD_MATH_COMMA ,
    #endif
    #define MATH_GLUE(a,b) a##B
    #define MATH_STRCAT(a,b) MATH_GLUE(a,b)
    #define ADD_MATH_BASENAME(name)    \
        MATH_STRCAT(MATH_BASENAME(),name)
#endif        /* MATH_BASENAME */
```

Selectors.k.h (Cont.)

- This section defines the prototype of **MathAdd** for the dispatch helper:

```
#ifdef MATH_BASENAME
    EXTERN_API (ComponentResult)  \
        ADD_MATH_BASENAME(Add)  \
        (MATH_GLOBALS() ADD_MATH_COMMA      \
        SInt16 inFirstNum, SInt16 inSecondNum,      \
        SInt32 * outResult);
```

- This lines up with our definition in **MathComponentSelectors.h**

Selectors.k.h (Cont.)

- This is the **MathAdd** procedure info:

```
enum { uppMathAddProcInfo =
    kPascalStackBased |
    RESULT_SIZE(SIZE_CODE
        (sizeof(ComponentResult))) |
    STACK_ROUTINE_PARAMETER(1,
        SIZE_CODE(sizeof(ComponentInstance))) |
    STACK_ROUTINE_PARAMETER(2,
        SIZE_CODE(sizeof(SInt16))) |
    STACK_ROUTINE_PARAMETER(3,
        SIZE_CODE(sizeof(SInt16))) |
    STACK_ROUTINE_PARAMETER(4,
        SIZE_CODE(sizeof(SInt32 *))) };
```

Let’s Write the Routines . . .

- We will look at how each of the routines in our component is implemented

The Open Routine

```
EXTERN_API (ComponentResult) __MathOpen
    (MathGlobalsPtr globals, ComponentInstance self)
{
    ComponentResult         result;

    result = noErr;
    globals = (MathGlobalsPtr) NewPtrClear (sizeof
        (MathGlobals));
    if (globals != NULL) {
        SetComponentInstanceStorage (self, (Handle)
            globals);
        globals->self = globals->target = self;
    } else result = MemError();
    return (result);
}
```

The Close Routine

```
EXTERN_API (ComponentResult) __MathClose
    (MathGlobalsPtr globals, ComponentInstance self)
{
    ComponentResult        result;

    result = noErr;
    if (globals != NULL)
        DisposePtr ((Ptr) globals);

    return (result);
}
```

The Version Routine

```
EXTERN_API (ComponentResult) __MathVersion
    (MathGlobalsPtr globals)
{
    globals;    /* Suppress “unused variable” warning */

    return (kMathAPIVersion |
        kMathImplementationVersion);
}
```

The CanDo Routine

- <insert Jamba Juice run here>
- We do not have to write this routine—the dispatch helper generates it for us automatically

The Register Routine

```
EXTERN_API (ComponentResult) __MathRegister
    (MathGlobalsPtr globals)
{
    SInt32  response;
    OSErr       result;
    globals;    /* Suppress “unused variable” warning */
    response = 0L;
    result = Gestalt (gestaltQuickTimeVersion,
        &response);
    if ((result == noErr) && (response >= 0x04128000))
        return (noErr);
    else
        return (-1L);
}
```

The Target Routine

```
EXTERN_API (ComponentResult) __MathTarget
    (MathGlobalsPtr globals, ComponentInstance target)
{
    globals;    /* Suppress “unused variable” warning */

    if (target == NULL)
        globals->target = globals->self;
    else
        globals->target = target;

    return (noErr);
}
```

The Unregister Routine

```
EXTERN_API (ComponentResult) __MathUnregister
    (MathGlobalsPtr globals)
{
    globals;    /* Suppress “unused variable” warning */

    /* Perform any cleanup necessary to allow this
      component to be unregistered */

    return (noErr);
}
```

Public Resources

- The Component Manager provides for public and private resources
- Private resources are those intended for use solely by the component implementation
- Public resources are those intended for use by other software as well as the component implementation

Public Resources (Cont.)

- A component resource map (‘**thnr**’) resource is used to associate a component’s private resource type and ID with a public type and ID
- The **GetComponentPublicResource** routine is used to access public resources

Public Resources (Cont.)

- Your component does not have to implement the **GetPublicResource** routine in order for your component to export public resources—only the ‘**thnr**’ resource is needed

The MathAdd Routine

```
EXTERN_API (ComponentResult) __MathAdd
    (MathGlobalsPtr globals, SInt16 inFirstNum,
    SInt16 inSecondNum, SInt32 * outResult)
{
    ComponentResult         result;

    globals;   /* Suppress “unused variable” warning */
    result = noErr;
    if (outResult != NULL)
        *outResult = inFirstNum + inSecondNum;
    else
        result = paramErr;
    return (result);
}
```

Component Glue

- We need to write some “glue” to help the Component Manager do the right thing when a client calls our routine
- We will do this by creating a stub library that both our component and any of our component’s clients link against
- We will also define a couple of glue helper routines—**CallMacComponent** and **CallWinComponent**

CallMacComponent

```
#if TARGET_API_MAC_OS8
    enum { uppCallComponentProcInfo =
        kPascalStackBased |
        RESULT_SIZE(SIZE_CODE
        (sizeof(ComponentResult))) |
        STACK_ROUTINE_PARAMETER(1,
        kFourByteCode) };
    #define CallMacComponent(gluePB)            \
        CallUniversalProc (CallComponentUPP,    \
            uppCallComponentProcInfo, &gluePB)
#else
    #define CallMacComponent(gluePB)            \
        CallComponentDispatch                          \
            ((ComponentParameters *)&gluePB)
#endif
```

CallWinComponent

```
ComponentResult
    CallWinComponentWithThreeParams
    (ComponentInstance ci, UInt8 flags, SInt16 what,
    SInt32 param1, SInt32 param2, SInt32 param3) {
        union { ComponentParameters comp;
            SInt8 dummy (sizeof (ComponentParameters)
            + (3 * sizeof (SInt32))); } CompParams;
    CompParams.comp.flags = flags;
    CompParams.comp.paramSize =
        sizeof (CompParams.dummy);
    CompParams.comp.what = what;
    CompParams.comp.params[0] = param1;
    CompParams.comp.params[1] = param2;
    CompParams.comp.params[2] = param3;
    return (CallComponent (ci, &CompParams.comp)); }
```

MathComponentGlue

```
EXTERN_API (ComponentResult) MathAdd
    (ComponentInstance ci, SInt16 inFirstNum,
    SInt16 inSecondNum, SInt32 * outResult)
{
#if TARGET_OS_WIN32
    return (CallWinComponentWithThreeParams
        (ci, 0, kMathAddSelect, (SInt32) inFirstNum,
        (SInt32) inSecondNum, (SInt32) outResult));
#else
```

MathComponentGlue

```
#define kMathAddParamSize      \
    (sizeof (MathAddParams))

struct MathAddParams {
    SInt32 *        outResult;
    SInt16  inSecondNum;
    SInt16  inFirstNum; };
typedef struct MathAddParams MathAddParams;
struct MathAddGluePB {
    UInt8                         componentFlags;
    UInt8                         componentParamSize;
    SInt16                  componentWhat;
    MathAddParams                 params;
    ComponentInstance  inInstance; };
typedef struct MathAddGluePB MathAddGluePB;
```

MathComponentGlue

```
    MathAddGluePB  gluePB;

    gluePB.componentFlags = 0;
    gluePB.componentParamSize =
        kMathAddParamSize;
    gluePB.componentWhat = kMathAddSelect;
    gluePB.params.inFirstNum = inFirstNum;
    gluePB.params.inSecondNum = inSecondNum;
    gluePB.params.outResult = outResult;
    gluePB.inInstance = ci;
    return (CallMacComponent (gluePB));
#endif
}
```

What the Heck Was That All About?

- Basically, we are building up a **ComponentParameters** structure (refer to **Components.h**) so that the ComponentManager receives data in a format it expects before it passes the data along to a component function

Windows Entry Point

- A Windows component is basically a DLL, so it needs a main entry point:

```
static HINSTANCE ghInst = NULL;
BOOL WINAPI DllMain (HANDLE hInst,
    ULONG ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        case DLL_PROCESS_DETACH:
            break; }
    return TRUE; }
```

Module Definition

- A Windows module definition is used to tell the linker where to find the component’s main entry point
- This is usually in a module definition (.def) file

```
LIBRARY MathComponent

EXPORTS

    DllMain    @1
    __MathComponentDispatch
```

PowerPC ‘thng’

```
resource ‘thng’ (kTheComponentResID) {
    kTheComponentType, kTheComponentSubType,
    kTheComponentManufacturer, 0,
    kAnyComponentFlagsMask, 0, 0,
    ‘strn’, kTheComponentResID,
    ‘stri’, kTheComponentResID, 0, 0,
    (kTheComponentAPIVersion |
    kTheComponentImplementationVersion),
    componentDoAutoVersion |
    componentWantsUnregister |
    componentHasMultiplePlatforms, 0, 0,
    { cmpWantsRegisterMessage, ‘_PPC’,
    kTheComponentResID,
    platformPowerPC },
    ‘thnr’, kTheComponentResID };
```

Carbon CFM ‘thng’

```
resource ‘thng’ (kTheComponentResID) {
    kTheComponentType, kTheComponentSubType,
    kTheComponentManufacturer, 0,
    kAnyComponentFlagsMask, 0, 0,
    ‘strn’, kTheComponentResID,
    ‘stri’, kTheComponentResID, 0, 0,
    (kTheComponentAPIVersion |
    kTheComponentImplementationVersion),
    componentDoAutoVersion |
    componentWantsUnregister |
    componentHasMultiplePlatforms, 0, 0,
    { cmpWantsRegisterMessage, ‘cfrg’,
    kTheComponentResID,
    platformPowerPCNativeEntryPoint},
    ‘thnr’, kTheComponentResID };
```

Carbon Mach-O ‘thng’

```
resource ‘thng’ (kTheComponentResID) {
    kTheComponentType, kTheComponentSubType,
    kTheComponentManufacturer, 0,
    kAnyComponentFlagsMask, 0, 0,
    ‘strn’, kTheComponentResID,
    ‘stri’, kTheComponentResID, 0, 0,
    (kTheComponentAPIVersion |
    kTheComponentImplementationVersion),
    componentDoAutoVersion |
    componentWantsUnregister |
    componentHasMultiplePlatforms, 0, 0,
    { cmpWantsRegisterMessage, ‘dlle’,
    kTheComponentResID,
    platformPowerPCNativeEntryPoint},
    ‘thnr’, kTheComponentResID };
```

Windows ‘thng’

```
resource ‘thng’ (kTheComponentResID) {
    kTheComponentType, kTheComponentSubType,
    kTheComponentManufacturer, 0,
    kAnyComponentFlagsMask, 0, 0,
    ‘strn’, kTheComponentResID,
    ‘stri’, kTheComponentResID, 0, 0,
    (kTheComponentAPIVersion |
    kTheComponentImplementationVersion),
    componentDoAutoVersion |
    componentWantsUnregister |
    componentHasMultiplePlatforms, 0, 0,
    { cmpWantsRegisterMessage, ‘dlle’,
    kTheComponentResID,
    platformWin32},
    ‘thnr’, kTheComponentResID };
```

The ‘cfrg’ Resource

```
resource ‘cfrg’ (0) { {
    extendedEntry {
        kPowerPCCFragArch, kIsCompleteCFrag,
        kNoVersionNum, kNoVersionNum,
        kDefaultStackSize, kNoAppSubFolder,
        kImportLibraryCFrag,
        kDataForkCFragLocator,
        kZeroOffset, kCFragGoesToEOF,
        “Math”, ‘cpnt’,
        “\0x00\0x80”,
        “”, “”, “Math” };
    };
};
```

The ‘dlle’ Resource

```
resource ‘dlle’ (kTheComponentResID) {
        “__MathComponentDispatch”
};
```

‘strn’ and ‘stri’ Resources

```
resource ‘strn’ (kTheComponentResID)
{
	“Math”
};

resource ‘stri’ (kTheComponentResID)
{
	“Does basic math operations.”
};
```

The ‘thnr’ Resource

```
resource ‘thnr’ (kTheComponentResID)
{
    {
        ‘STR ‘, 1, 0,
        ‘strn’, kTheComponentResID, 0
    }
}
```

Rez and RezWack

- **Rez** is a DOS console application used to compile Mac OS resource (**.r**) files into Windows compatible resource fork (**.qtr**) files
- **RezWack** is a DOS console application used to embed a resource fork (**.qtr**) file into a Windows DLL or executable file

Registering Components

- There are three main methods for registering components:
  - Auto registration
  - Reinstaller 3
  - Programmatic registration

Auto Registration

- Mac OS 7.x–9.x: Copy component into System Folder : Extensions : QuickTime Extensions and reboot
- Mac OS X: Copy component into /Library/QuickTime or /Users/<username>/Library/QuickTime
- Windows: Copy component into Windows/System32/QuickTime or into application directory

Reinstaller 3

- Reinstaller 3 is a utility to register components in Mac OS 7.x–9.x without rebooting
- Download at **http://developer.apple.com/quicktime/quicktimeintro/tools.index.html**

Programmatic Registration

- **RegisterComponent** registers a component given its description and main entry point
- **RegisterComponentResource** registers a component given its component resource

Why Use It?

- Programmatic registration is handy if you do not want other applications to have access to your component
- It is useful as an alternate means of source-level debugging your component

Debugging Components

- Source level debugging is available in all major development environments
- On Mac OS 7.x–9.x, the ‘**thng**’ MacsBug **dcmd** is very helpful
- On Windows, you can use the Visual Studio editor or **RezDet** to make sure your component resource was properly attached to your component

Debugging Components

- **DebugStr** (Mac OS) and **OutputDebugStr** (Windows) are your friends
- Use them to determine whether your component's routines are being invoked, and in what order

How Do I Deliver My Component?

- You can ship it yourself, either alone or with an application that uses it
- You can apply to Apple's QuickTime Component Download Program at **http://developer.apple.com/quicktime/qtcdform.html**

Common Problems

- Problem: My component compiles and links, but it is never registered
- If you have a Register routine, make sure it is not failing
- If you are auto-registering, make sure your component is installed in the correct directory

Common Problems

- For Mac OS X and Windows, make sure your ‘**dlle**’ resource is defined
- For Carbon, make sure your ‘**cfrg**’ resource is defined
- Make sure there is no mismatch between your dispatcher name and the name exported by your component

Common Problems

- Problem: My custom component's routine has wacky values in its arguments, and sometimes crashes
- If this is your own component type, make sure the sizes of all arguments in macros, **enums**, and declarations are correct—otherwise, arguments on the stack can be misaligned

Common Problems

- Problem: My component's Register routine never gets called
- Verify "**cmpWantsRegisterMessage**" flag is set in the component platform entry of your component resource, not in the 68K flags field at the head of the resource, and not in the **componentRegistrationFlags** field

Common Problems

- If you are programatically registering your component, you will need to call **CallComponentRegister** to force your component's Register routine to execute

Common Problems

- Problem: My component's routine never gets called
- Make sure your component dispatch file is properly accounting for all selectors

Common Problems

- Problem: My component opens another of my components and needs to share internal state with it
- You can create a private selector in your component API that allows the opening component to call the opened component with the state information

Common Problems

- Problem: My component requires a connection to hardware resource “x”, and “x” only supports a single connection per physical device, although multiple physical devices can be present; How should my component handle this?
- Wow, that is a good one

Common Problems

- Register your component once for each unique physical device, and allow only a single instance of each component to be open at any time
- Assumes that each physical device can be uniquely and persistently identified, and that it is possible to track device connection status across address spaces

Common Problems

- Register your component once, and allow multiple instances of your component, up to the number of physical devices present
- Has same qualifications as the former approach, and also makes it slightly more difficult to count the devices using the Component Manager

Common Problems

- Register your component once, with no limitations on how many instances are allowed
- Has same qualifications as the former approaches, and also makes device management more cumbersome, particularly since a single hardware connection can be shared among multiple clients

Common Problems

- There is no magic answer to this issue
- You will have to experiment to see what works best for your situation

Closing Advice

- Do not allow multiple instances of your component if it can not support multiple instances
- Be sure to clean up properly when an instance of your component is closed
- Use Gestalt to make sure a QuickTime service your component needs is present before trying to use it

Closing Advice

- Use the **ComponentDispatchHelper**
- Do not forget the Component Manager has platform differences
- Look for an existing component API before rolling your own—QuickTime's got lots of them
- Subscribe to the QuickTime-API mailing list—do not forget to contribute!

QuickTime Roadmap

600 The State of QuickTime in 2002	Room A2 **Wed., 9:00am**
601 Building QuickTime Savvy Apps	Room A2 **Wed., 10:30am**
602 QuickTime for Video-Intensive Applications	Room A2 **Wed., 2:00pm**
603 Media Integration with QuickTime	Room A2 **Wed., 3:30pm**
604 Delivering Content via Interactive QuickTime	Room A2 **Wed., 5:00pm**

QuickTime Roadmap

FF010 QuickTime	Room J1 **Fri., 10:30am**
606 QuickTime for the Web	Room A2 **Fri., 2:00pm**
607 QuickTime and MPEG4: A Technical Overview	Room A2 **Fri., 3:30pm**

Who to Contact

Jeff Lowe
QuickTime Evangelist
jefflowe@apple.com

Gary Woodcock
Discreet, Inc.
garyw@unthinkable.com

For More Information

- The QuickTime Developer Series published by Morgan Kaufmann
- QuickTime Developer Web Site
  http://developer.apple.com/quicktime
- QuickTime-API Mailing List
  http://lists.apple.com/mailman/listinfo/quicktime-api
- Download the sample code at
  http://www.unthinkable.com/downloads/wwdc2002/
  - the files are math.sit.hqx or math.zip

Reminder

The QuickTime Engineering Team
Is Holding a “Hands On Lab” Everyday
from 1:00-4:00pm in Room G . . Stop By!

WWDC2002

WWDC2002

WWDC2002